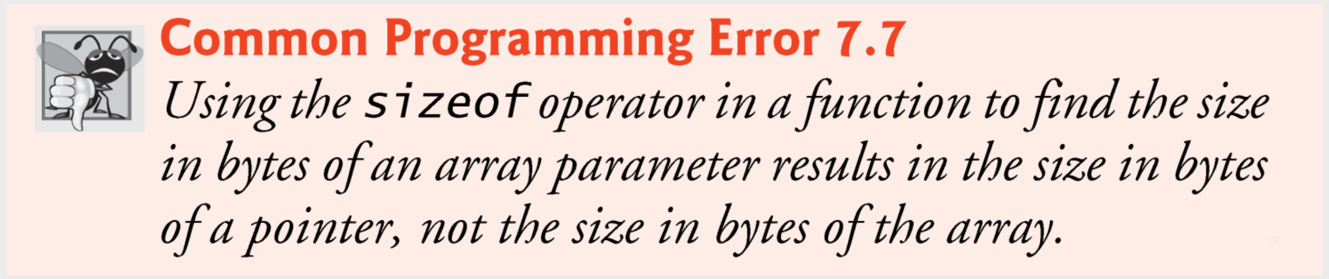
Pointers Part II

C++ How to Program, Late Objects Version, 7/e

Dr. Jian-Ren Hou

7.7 sizeof Operator

- The unary operator sizeof determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation.
- When applied to the name of an array, the sizeof operator returns the total number of bytes in the array as a value of type size_t.
- When applied to a pointer parameter in a function that receives an array as an argument, the sizeof operator returns the size of the pointer in bytes—not the size of the array.



Common Programming Error 7.7

Using the `sizeof` *operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.*

```cpp
// Fig. 7.14: fig07_14.cpp
// Sizeof operator when used on an array name
// returns the number of bytes in the array.
#include <iostream>
using namespace std;

size_t getSize( double * ); // prototype

int main()
{
   double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system

   cout << "The number of bytes in the array is " << sizeof( array );

   cout << "\nThe number of bytes returned by getSize is "
      << getSize( array ) << endl;
} // end main

// return size of ptr
size_t getSize( double *ptr )
{
   return sizeof( ptr );
} // end function getSize
```

Fig. 7.14 | sizeof operator when applied to an array name returns the number of bytes in the array. (Part 1 of 2.)

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 7.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 2 of 2.)

7.7 sizeof Operator (cont.)

- The number of elements in an array also can be determined using the results of two sizeof operations.
- Consider the following array declaration:
    - double realArray[22];
- To determine the number of elements in the array, the following expression (which is evaluated at compile time) can be used:
    - sizeof realArray / sizeof(realArray[0])
- The expression determines the number of bytes in array realArray and divides that value by the number of bytes used in memory to store the array’s first element.

7.7 sizeof Operator (cont.)

- Figure 7.15 uses sizeof to calculate the number of bytes used to store most of the standard data types.
- The output shows that the types double and long double have the same size.
  - Types may have different sizes based on the platform running the program.

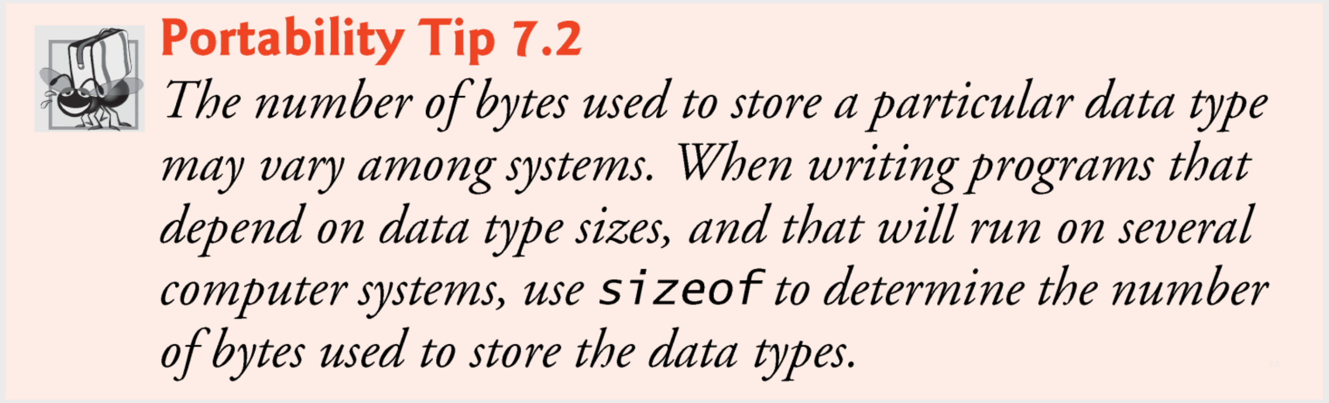
```cpp
// Fig. 7.15: fig07_15.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using namespace std;

int main()
{
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[ 20 ]; // array of int
   int *ptr = array; // variable of type int *

   cout << "sizeof c = " << sizeof c
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof s
      << "\tsizeof(short) = " << sizeof( short )
      << "\nsizeof i = " << sizeof i
```

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 1 of 2.)

```cpp
         << "\tsizeof(int) = " << sizeof( int )
         << "\nsizeof l = " << sizeof l
         << "\tsizeof(long) = " << sizeof( long )
         << "\nsizeof f = " << sizeof f
         << "\tsizeof(float) = " << sizeof( float )
         << "\nsizeof d = " << sizeof d
         << "\tsizeof(double) = " << sizeof( double )
         << "\nsizeof ld = " << sizeof ld
         << "\tsizeof(long double) = " << sizeof( long double )
         << "\nsizeof array = " << sizeof array
         << "\nsizeof ptr = " << sizeof ptr << endl;
} // end main
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 2 of 2.)



Portability Tip 7.2

The number of bytes used to store a particular data type may vary among systems. When writing programs that depend on data type sizes, and that will run on several computer systems, use `sizeof` *to determine the number of bytes used to store the data types.*

In-Class exercise

- It is a standard trick that to calculate the size of an array A by
  - int A[3]={};
  - sizeof(A)/sizeof(A[0]);

1. Please try the trick in your main().
2. What will happen if you pass an array to a function to calculate the size of the array via the trick above?

```
Ex: void getSize(int B[])
{cout << sizeof(B)/sizeof(B[0]) << endl ;}
```

7.7 sizeof Operator (cont.)

- Operator sizeof can be applied to any expression or type name.
- When sizeof is applied to a variable name (which is not an array name) or other expression, the number of bytes used to store the specific type of the expression’s value is returned.
- The parentheses used with sizeof are required only if a type name is supplied as its operand.

Common Programming Error 7.8

Omitting the parentheses in a `sizeof` *operation when the operand is a type name is a compilation error.*

Performance Tip 7.2

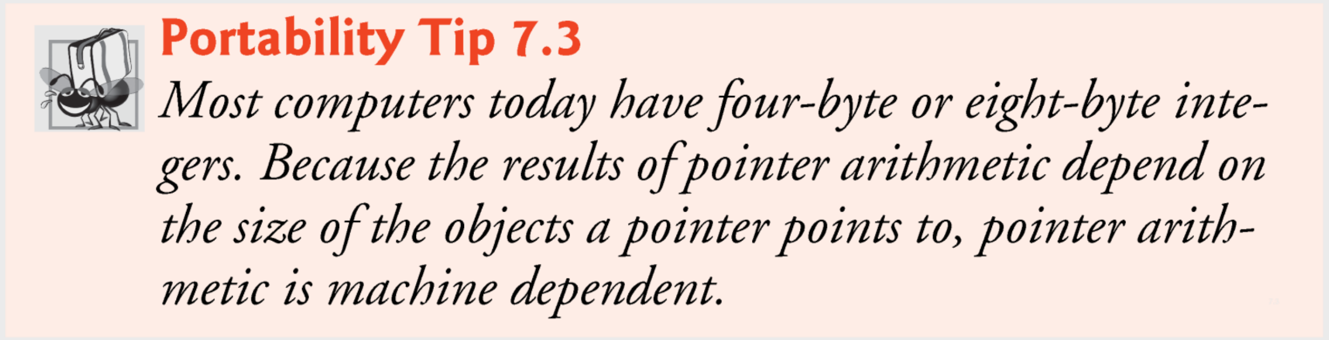
Because `sizeof` *is a compile-time unary operator, not an execution-time operator, using* `sizeof` *does not negatively impact execution performance.*

Error-Prevention Tip 7.3

To avoid errors associated with omitting the parentheses around the operand of operator `sizeof`*, include parentheses around every* `sizeof` *operand.*

7.8 Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- pointer arithmetic—certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type



Portability Tip 7.3

Most computers today have four-byte or eight-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Assume that array int v[5] has been declared and that its first element is at memory location 3000.
- Assume that pointer vPtr has been initialized to point to v[0] (i.e., the value of vPtr is 3000).
- Figure 7.16 diagrams this situation for a machine with four-byte integers.

7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- In conventional arithmetic, the addition 3000 + 2 yields the value 3002.
  - This is normally not the case with pointer arithmetic.
  - When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers.
  - The number of bytes depends on the object's data type.

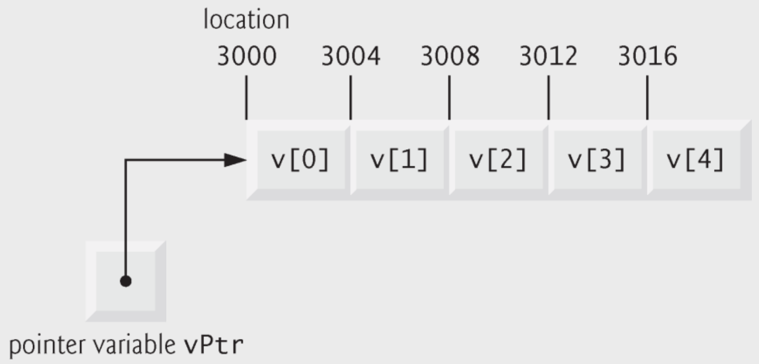


Fig. 7.16 | Array v and a pointer variable int *vPtr that points to v.

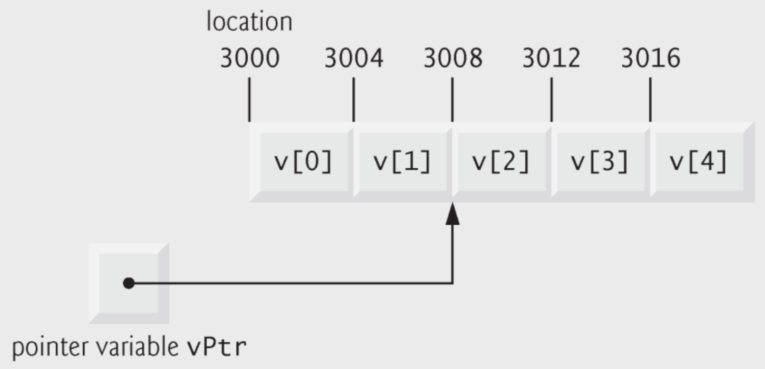
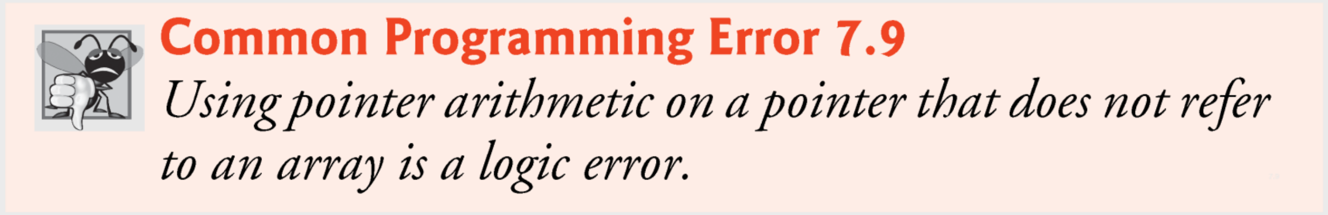


Fig. 7.17 | Pointer `vPtr` after pointer arithmetic.

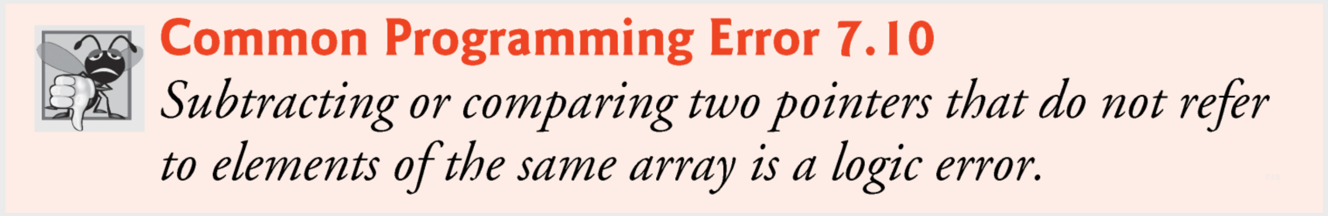
7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Pointer variables pointing to the same array may be subtracted from one another.
- For example, if vPtr contains the address 3000 and v2Ptr contains the address 3008, the statement
  - x = v2Ptr - vPtr;
- would assign to x the number of array elements from vPtr to v2Ptr—in this case, 2.
- Pointer arithmetic is meaningless unless performed on a pointer that points to an array.



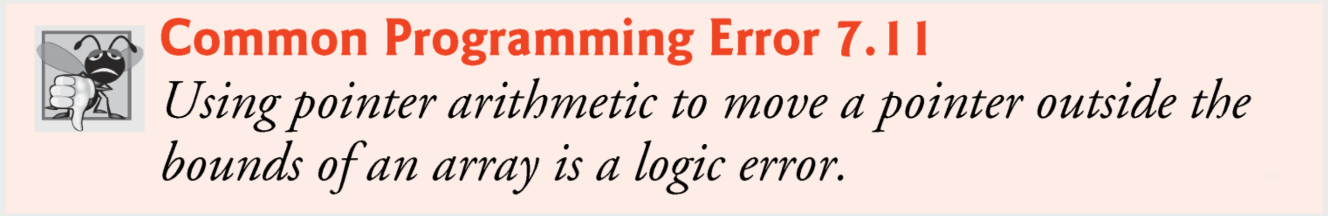
Common Programming Error 7.9

Using pointer arithmetic on a pointer that does not refer to an array is a logic error.



Common Programming Error 7.10

Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.



Common Programming Error 7.11

Using pointer arithmetic to move a pointer outside the bounds of an array is a logic error.

7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- A pointer can be assigned to another pointer if both pointers are of the same type.

- All pointer types can be assigned to a pointer of type void * without casting.

Common Programming Error 7.12

Assigning a pointer of one type to a pointer of another (other than `void *`*) without using a cast (normally a* `reinterpret_cast`*) is a compilation error.*

7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- A void * pointer cannot be dereferenced.
  - The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to void, this number of bytes cannot be determined from the type.

7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Pointers can be compared using equality and relational operators.
  - Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
  - Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).

7.9 Relationship Between Pointers and Arrays

- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array subscripting.
- Assume the following declarations:
  - int b[5]; // create 5-element int array b
    int *bPtr; // create int pointer bPtr
- Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set bPtr to the address of the first element in array b with the statement
  - bPtr = b; // assign address of array b to bPtr
- equivalent to
  - bPtr = &b[0]; // also assigns address of array b to bPtr

7.9 Relationship Between Pointers and Arrays (cont.)

- Array element b[3] can alternatively be referenced with the pointer expression
    - *(bPtr + 3)
- The 3 in the preceding expression is the offset to the pointer.
- This notation is referred to as pointer/offset notation.
    - The parentheses are necessary, because the precedence of * is higher than that of +.

7.9 Relationship Between Pointers and Arrays (cont.)

- Just as the array element can be referenced with a pointer expression, the address
  - &b[3]
- can be written with the pointer expression
  - bPtr + 3
- The array name (which is implicitly const) can be treated as a pointer and used in pointer arithmetic.
- For example, the expression
  - *(b + 3)
- also refers to the array element b[3].
- In general, all subscripted array expressions can be written with a pointer and an offset.

Common Programming Error 7.14

Although array names are pointers to the beginning of the array, array names cannot be modified in arithmetic expressions, because array names are constant pointers.

7.9 Relationship Between Pointers and Arrays (cont.)

- Pointers can be subscripted exactly as arrays can.
- For example, the expression
    - bPtr[1]
- refers to the array element b[1]; this expression uses pointer/subscript notation.

7.9 Relationship Between Pointers and Arrays (cont.)

- Figure 7.18 uses the four notations discussed in this section for referring to array elements—array subscript notation, pointer/offset notation with the array name as a pointer, pointer subscript notation and pointer/offset notation with a pointer—to accomplish the same task, namely printing the four elements of the integer array b.

```cpp
// Fig. 7.18: fig07_18.cpp
// Using subscripting and pointer notations with arrays.
#include <iostream>
using namespace std;

int main()
{
   int b[] = { 10, 20, 30, 40 }; // create 4-element array b
   int *bPtr = b; // set bPtr to point to array b

   // output array b using array subscript notation
   cout << "Array b printed with:\n\nArray subscript notation\n";

   for ( int i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   // output array b using the array name and pointer/offset notation
   cout << "\nPointer/offset notation where "
      << "the pointer is the array name\n";

   for ( int offset1 = 0; offset1 < 4; offset1++ )
      cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)

```cpp

    // output array b using bPtr and array subscript notation
    cout << "\nPointer subscript notation\n";

    for ( int j = 0; j < 4; j++ )
       cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

    cout << "\nPointer/offset notation\n";

    // output array b using bPtr and pointer/offset notation
    for ( int offset2 = 0; offset2 < 4; offset2++ )
       cout << "*(bPtr + " << offset2 << ") = "
          << *( bPtr + offset2 ) << '\n';
} // end main
```

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)

```
Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 3 of 3.)

7.10 Pointer-Based String Processing

- This section introduces C-style, pointer-based strings.
- C++’s string class is preferred for use in new programs, because it eliminates many of the security problems that can be caused by manipulating C strings.
- We cover C strings here for a deeper understanding of arrays.
- Also, if you work with legacy C++ programs, you may be required to manipulate these pointer-based strings.

7.10 Pointer-Based String Processing (cont.)

- Characters are the fundamental building blocks of C++ source programs.
- Character constant
  - An integer value represented as a character in single quotes.
  - The value of a character constant is the integer value of the character in the machine's character set.
- A string is a series of characters treated as a single unit.
  - May include letters, digits and various special characters such as +, -, *, /and $.
- String literals, or string constants, in C++ are written in double quotation marks
- A pointer-based string is an array of characters ending with a null character ('\0').
- A string is accessed via a pointer to its first character.

Common Programming Error 7.15

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.

Common Programming Error 7.16

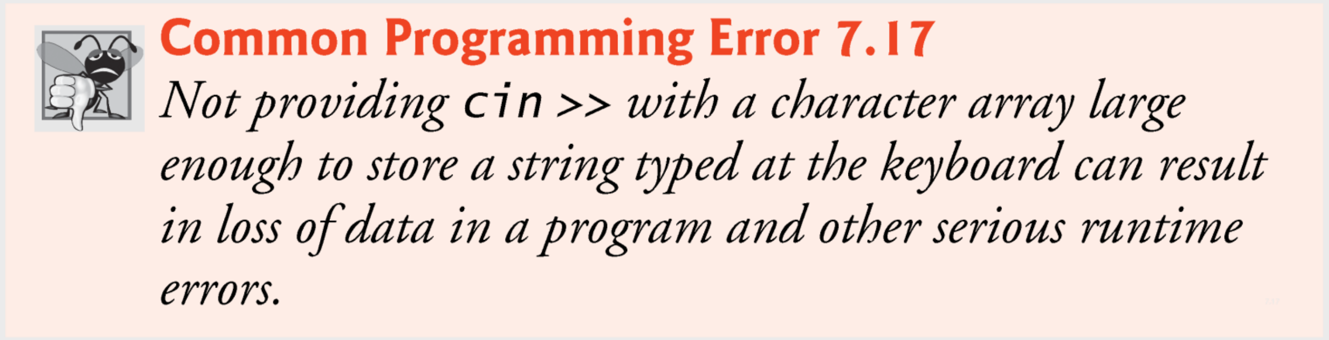
Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.

Error-Prevention Tip 7.4

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.

7.10 Pointer-Based String Processing (cont.)

- The value of a string literal is the address of its first character, but the sizeof a string literal is the length of the string including the terminating null character.
- A string literal may be used as an initializer in the declaration of either a character array or a variable of type char *.
- String literals have static storage class (they exist for the duration of the program) and may or may not be shared if the same string literal is referenced from multiple locations in a program.
- The effect modifying a string literal is undefined; thus, you should always declare a pointer to a string literal as const char *.
- When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character.



Common Programming Error 7.17

Not providing `cin >>` *with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other serious runtime errors.*

7.10 Pointer-Based String Processing (cont.)

- Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation.
- A string can be read into a character array using stream extraction with cin.
- The setw stream manipulator can be used to ensure that the string read into word does not exceed the size of the array.
  - Applies only to the next value being input.

7.10 Pointer-Based String Processing (cont.)

- In some cases, it’s desirable to input an entire line of text into a character array.
- For this purpose, the cin object provides the member function getline.
- Three arguments—a character array in which the line of text will be stored, a length and a delimiter character.
- The function stops reading characters when the delimiter character '\n' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument.
- The third argument to cin.getline has '\n' as a default value.

In-Class exercise

- Given

char A_name[]=“John Doe”;

Can you use array manipulation to output “John Goe”?

7.11 Arrays of Pointers

- Arrays may contain pointers.
- A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a string array.
- Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string.
- const char * const suit[4] =
    { "Hearts", "Diamonds",
      "Clubs", "Spades" };
  - An array of four elements.
  - Each element is of type “pointer to char constant data.”

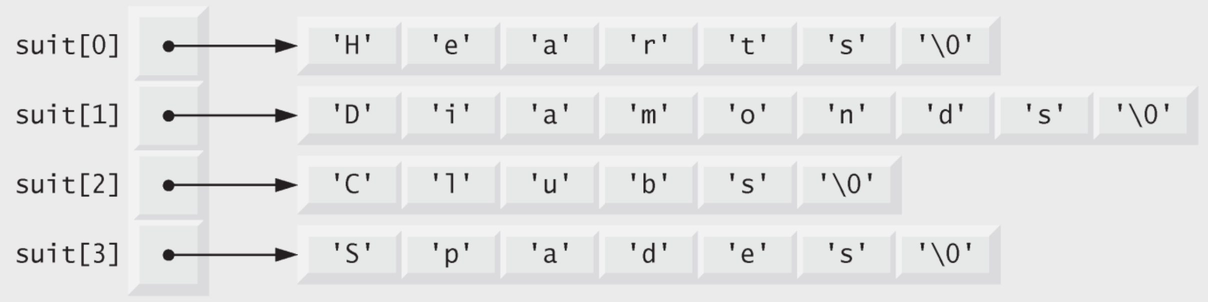

suit[0]
'H' 'e' 'a' 'r' 't' 's' '\0'
suit[1]
'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'
suit[2]
'C' 'l' 'u' 'b' 's' '\0'
suit[3]
'S' 'p' 'a' 'd' 'e' 's' '\0'

Fig. 7.19 | Graphical representation of the suit array.